PROCESSOR AND METHOD OF EXECUTING A LOAD INSTRUCTION THAT BIFURCATE LOAD EXECUTION INTO TWO OPERATIONS

BACKGROUND OF THE INVENTION

1. Technical Field:

The present invention relates in general to data processing and, in particular, to a processor and method of performing load operations in a processor. Still more particularly, the present invention relates to a processor and method of processing a load instruction that bifurcate load execution into two separate operations.

2. Description of the Related Art:

Most processors' instruction set architectures (ISAs) include a load or similar type of instruction that, when executed, causes the processor to load specified data from memory (e.g., cache memory or system memory) into the processor's internal registers. Conventional processors handle the execution of load instructions in one of two ways. First, a processor may execute load instructions strictly in program order. In general, the execution of load instructions with strict adherence to program order is viewed as disadvantageous given the fact that at least some percentage of data specified by load instructions will not be present in the processor's cache. In such cases, the processor must stall the execution of the instructions following the load until the data specified by the load is retrieved from memory.

Alternatively, a processor may permit load instructions to execute out-of-order with respect to the

programmed sequence of instructions. In general, out-of-order execution of load instructions is viewed as advantageous since operands required for execution are obtained from memory as soon as possible, thereby improving overall processor throughput. However, supporting out-of-order execution of load instructions entails additional complexity in the processor's architecture since, to guarantee correctness, the processor must be able to detect and cancel an out-of-order load instruction that loads data from a memory location targeted by a later-executed store instruction (executed in the same or a remote processor) preceding the load instruction in program order.

SUMMARY OF THE INVENTION

The present invention addresses the poor performance associated with in-order processors and eliminates much of the complexity associated with out-of-order machines by providing an improved processor and method of executing load instructions.

In accordance with the present invention, a processor implementing an improved method for executing load instructions includes execution circuitry, a plurality of registers, and instruction processing circuitry. The instruction processing circuitry fetches a load instruction and a preceding instruction that precedes the load instruction in program order, and in response to detecting the load instruction, translates the load instruction into separately executable prefetch and register operations. The execution circuitry performs at least the prefetch operation out-of-order with respect to the preceding instruction to prefetch data into the processor and subsequently separately executes the register operation to place the data into a register specified by the load instruction. In an embodiment in which the processor is an in-order machine, the register operation is performed in-order with respect to the preceding instruction.

All objects, features, and advantages of the present invention will become apparent in the following detailed written description.

BRIEF DESCRIPTION OF THE DRAWINGS

The novel features believed characteristic of the invention are set forth in the appended claims. The invention itself however, as well as a preferred mode of use, further objects and advantages thereof, will best be understood by reference to the following detailed description of an illustrative embodiment when read in conjunction with the accompanying drawings, wherein:

Figure 1 depicts an illustrative embodiment of a data processing system with which the method and system of the present invention may advantageously be utilized;

Figure 2A and **2B** illustrate two alternative embodiments of the translation of UISA load instructions into separately executable PREFETCH and REGISTER operations in accordance with the present invention; and

Figure 3 is an exemplary load data queue that may be utilized to temporarily buffer load data in accordance with the present invention.

DETAILED DESCRIPTION OF ILLUSTRATIVE EMBODIMENT

With reference now to the figures and in particular with reference to **Figure 1**, there is illustrated a block diagram of an exemplary embodiment of a data processing system with which the present invention may advantageously be utilized. As shown, the data processing system includes at least one processor, indicated generally at **10**, which, as discussed further below, includes various execution units, registers, buffers, memories, and other functional units that are all formed within a single integrated circuit. Processor **10** is coupled by a bus interface unit (BIU) **14** to a bus **12** and other components of the data processing system, such as system memory **8** or a second processor **10** (not illustrated).

Processor **10** includes an on-chip multi-level cache hierarchy **16** that provides low latency access to cache lines of instructions and data that correspond to memory locations in system memory **8**. In the depicted embodiment, cache hierarchy **16** includes separate level one (L1) instruction and data caches **13** and **15** and a unified level two (L2) cache **17**. An instruction sequencing unit (ISU) **20** requests instructions from cache hierarchy **16** by supplying effective addresses (EAs) of cache lines of instructions. In response to receipt of an instruction request, cache hierarchy **16** translates the provided EA into a real address and outputs the specified cache line of instructions to instruction translation unit **18**. Instruction translation unit **18** then translates each cache line of instructions from a user instruction

set architecture (UISA) into a possibly different number of internal ISA (IISA) instructions that are directly executable by the execution units of processor **10**. The instruction translation may be performed, for example, by reference to microcode stored in a read-only memory (ROM) template. In at least some embodiments, the UISA-to-IISA translation results in a different number of IISA instructions than UISA instructions and/or IISA instructions of different lengths than corresponding UISA instructions.

Following instruction translation by ITU **18**, ISU **20** temporarily buffers the IISA instructions until the instructions can be dispatched to one of the execution units of processor **10** for execution. In the illustrated embodiment, the execution units of processor **10** include integer units (IUs) **24** for executing integer instructions, a load-store unit (LSU) **26** for executing load and store instructions, and a floating-point unit (FPU) **28** for executing floating-point instructions. Each of execution units **24-28** is preferably implemented as an execution pipeline having a number of pipeline stages.

During execution within one of execution units **24-28**, an instruction receives operands (if any) from, and stores data results (if any) to one or more registers within a register file coupled to the execution unit. For example, IUs **24** execute integer arithmetic and logic instructions by reference to general-purpose register (GPR) file **32**, and FPU **28** executes floating-point arithmetic and logic instructions by reference to floating-point register (FPR) file **34**. LSU **26** executes load and store instructions to transfer data between

memory (e.g., cache hierarchy **16**) and either of GPR file **32** and FPR file **34**. After an execution unit finishes execution of an instruction, the execution unit notifies instruction sequencing unit **20**, which schedules completion of the instruction in program order. Upon completion of an instruction, the data results, if any, of the instruction form a portion of the architected state of processor **10**, and execution resources allocated to the instruction are made available for use in the execution of a subsequent instruction.

As noted above, much of the hardware and data flow complexity involved in processing load instructions in conventional processors is attributable to the execution of load and other instructions out-of-program order. In particular, the design philosophy of many conventional processors that permit out-of-order execution of instructions is to execute load instructions as early as possible to place specified data into a register file so that subsequent instructions having a dependency upon the load data are less likely to stall due to memory access latency. The processor must then detect data hazards (e.g., store instructions targeting the same address that are earlier in program order, but later in execution order) with respect to the data and discard the load data from the register file (and instructions executed utilizing that load data) in the event that the load data is found to be stale.

In accordance with the present invention, processor **10** simplifies the processing of UISA load instructions by translating at least some of these UISA load instructions into two separately executable IISA instructions. These two IISA instructions are defined

herein as a PREFETCH instruction that, if necessary, causes specified data to be prefetched from lower level memory (e.g., L2 cache **17** or system memory **8**)into L1 data cache **15** and a REGISTER instruction that transfers data specified by the UISA load instruction into a register file.

Referring now to **Figures 2A** and **2B**, there are depicted two alternative embodiments of the translation of UISA load instructions into separately executable PREFETCH and REGISTER instructions in accordance with the present invention. As illustrated in **Figure 2A**, in a first embodiment, ITU **18** translates UISA load instruction into two IISA LOAD instructions **40** and **42** that are identical except for the value of a register operation field **50**. Thus, while LOAD instructions **40** and **42** have matching opcode, register, and address fields **44, 46** and **48**, register field **50** of LOAD instruction **40** is reset to 0 to indicate a PREFETCH operation, and register field **50** of LOAD instruction **42** is set to 1 to indicate a REGISTER operation. A variation on this embodiment that could be implemented with or without instruction translation by ITU **18** would be for a single LOAD instruction to be supplied to ISU **20**, and for ISU **20** to issue the LOAD instruction twice for execution (e.g., from an instruction buffer) with differing settings of register field **50**.

Alternatively, as shown in **Figure 2B**, ITU **18** may translate a UISA load instruction into distinct IISA prefetch and register instructions **60** and **62**, respectively. As illustrated, IISA PREFETCH instruction

60 contains, in addition to an opcode field 64, at least a target address field 66 identifying operands that may be utilized to compute the memory address(es) from which load data is to be retrieved. IISA REGISTER instruction 62, by contrast, has a different opcode specified in its opcode field 64 and specifies in a register field 68 the register(s) into which the load data are to be transferred.

By translating UISA instructions to IISA instructions in this manner, memory access latency associated with load instructions can be masked as in complex out-of-order machines, even in processors of reduced complexity that execute instructions either in-order or only slightly out-of-order. As an example, an exemplary cache line of instructions fetched from cache hierarchy 16 may include the following UISA instructions:

ADD1
SUB1
MUL1
MUL2
ST
SUB2
LD
ADD2

where ADD1 is the earliest UISA instruction in program order, LD is a UISA load instruction, and ADD2 is the latest instruction in program order and is an addition instruction dependent upon the load data. According to the embodiment depicted in **Figure 2B**, these UISA instructions may be translated into the following sequence of IISA instructions:

ADD1
SUB1
MUL1
MUL2
ST

SUB2
PRE
REG
ADD2

where PRE and REG denote separately executable IISA PREFETCH and REGISTER instructions, respectively.

If instruction sequencing unit **20** enforces in-order execution, which is defined to mean that no instruction that changes the state of an architected register can be executed prior to an instruction preceding it in program order, processor **10** can still enjoy the chief benefits of executing load instructions out-of-order, that is, masking memory access latency, without the concomitant complexity by speculatively executing the IISA PREFETCH instruction prior to at least one instruction preceding it in program order. In this manner, cache hierarchy **16** can speculatively initiate prefetching of the load data into L1 data cache **15** to mask data access latency, while the REGISTER instruction (which alters the architected state of processor **10**) is still performed in-order. Table I summarizes an exemplary execution scenario, given the IISA instruction stream discussed above and an embodiment of processor **10** in which ISU **20** is capable of dispatching and retiring two instructions per cycle.

TABLE I

	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Cycle 6	Cycle 7
ADD1	D	X	C				
PRE	D	X		pre-fetch data to L1 data cache			
SUB1		D	X	C			
MUL1		D	X	C			
MUL2			D	X	C		
ST			D	X	C		
SUB2				D	X	C	
REG				D	X	C	
ADD2					D	X	C

In the exemplary scenario depicted in Table I, at the beginning of cycle 1, ISU **20** holds all nine of the IISA instructions, for example, in a deep instruction buffer that is preferably more than one cache line of instructions in depth. In response to detecting a PRE instruction available for dispatch in the instruction buffer, ISU **20** dispatches the PRE instruction out-of-order to LSU **26**, for example, concurrent with the dispatch of ADD1 to IU **24**.

During cycle 2, ISU **20** also decodes and dispatches the SUB1 and MUL1 instructions to IUs **24**. Meanwhile, IU **24** executes ADD1, and LSU **26** executes the PRE instruction to calculate a speculative effective address (EA) of the data to be loaded. This speculative

EA is then translated to a real address, for example, by reference to a conventional data translation lookaside buffer (TLB), and supplied to cache hierarchy **16** as a prefetch request. Thus, if the real address hits in L1 data cache **15**, then no further action is taken. However, if the real address misses in L1 data cache **15**, then the real address will be furnished to L2 cache **17** as a request address. In the event of a hit in L2 cache **17**, L2 cache **17** will load the associated data into L1 data cache **15**; however, if the real address misses in L2 cache **17**, then a request containing the real address will be sourced onto data bus **12** for servicing by system memory **18** or another processor **10**. Thus, execution of the PREFETCH instruction triggers prefetching of data into cache hierarchy **16** (and preferably L1 data cache **15**) that is likely to be loaded into a register file in response to execution of a REGISTER instruction. This prefetching is speculative, however, in that an intervening branch instruction may redirect the execution path, resulting in the REGISTER instruction not being executed. In addition, the contents of the registers utilized to compute the EA of the load data may be updated by an instruction executed between the PRE instruction and the associated REG instruction. However, because the PRE instruction merely affects the cache contents rather than the architected state of processor **10**, no corrective action need be taken in the event of mis-speculation.

Next, in cycle 3, ISU **20** completes the ADD1 instruction, and its result data become part of the architected state of processor **10**. As further shown in Table I, the SUB1 and MUL1 instructions are executed by

IUs **24**, and the MUL2 and ST instructions are decoded and dispatched to IU **24** and LSU **26**, respectively.

Assuming that the prefetch request missed in L1 data cache **15** and hit in L2 data cache **17**, during cycle 4 a copy of the prefetch data is loaded from L2 data cache **17** into L1 data cache **15**. The MUL2 and ST instructions are also executed by an IU **24** and LSU **26**, respectively. In addition, ISU **20** completes the SUB1 and MUL1 instructions and decodes and dispatches the SUB2 and REG instructions to an IU **24** and LSU **26**, respectively. Thus, as required by the in-order architecture of processor **10**, the REG instruction, which affects the architected state of processor **10** is dispatched, executed and completed no earlier than SUB2, the instruction preceding it in program order.

Next, in cycle 5, the MUL2 and ST instructions are completed by ISU **20**, and the SUB2 and REG instructions are executed by an IU **24** and LSU **26**, respectively. To execute the REG instruction, LSU **26** computes the EA of the load data and supplies the EA to cache hierarchy **16**, which translates the EA to a real address and determines whether the load data associated with that real address is resident in L1 data cache **15**. Because of the earlier speculative execution of the PRE instruction, in most cases the load data is resident in L1 data cache **15**, and the REG instruction can both execute and load data into one of register files **32** or **34** in the minimum data access latency permitted by cache hierarchy **16**, which in this case is a single cycle.

Thereafter, in cycle 6, the ADD2 instruction, which is dispatched in cycle 5, is executed by one of IUs **24** concurrent with the completion of the SUB2 and REG instructions by ISU **20**. As illustrated, because the PRE instruction speculatively prefetches the data required for the ADD2 instruction prior to execution of the REG instruction, the ADD2 instruction, which is dependent upon the load data, is permitted to execute without any latency. Finally, ISU **20** completes the ADD2 instruction during cycle 7.

It should be evident to those skilled in the art that various modifications of the exemplary processor described herein are possible and may be desirable, depending upon other architectural considerations. For example, it may be desirable for instruction translation unit **18** to be merged into ISU **20**. In addition, it may be desirable for a processor in accordance with the present invention to permit out-of-order execution of instructions other than memory access instructions (e.g., loads and stores), while requiring memory access instructions to be executed strictly in order. In general, permitting non-memory-access instructions to execute out-of-order would not introduce any additional complexity as compared to in-order execution since conventional in-order processors include logic for detecting and observing register data dependencies between instructions. Moreover, a processor in accordance with the present invention may chose to execute the PRE instruction by speculatively loading the data into buffer storage, rather than merely "priming" the cache hierarchy with a prefetch address. Buffering speculatively fetched load data in this manner is permitted even by in-order machines in that the content of the register files is not affected.

For example, **Figure 3** illustrates a load data queue **80** within LSU **26** that may be utilized to temporarily buffer load data received from cache hierarchy **16** in response to execution of a PREFETCH instruction. As shown, each entry of load data queue **80** associates load data retrieved from cache hierarchy **16** with the target address (TA) from which the load data was retrieved and the EA of the UISA load instruction, which is shared by and flows through processor **10** in conjunction with each of the PREFETCH and REGISTER IISA instructions. Thus, when LSU **26** subsequently executes a REG instruction, the EA of the UISA load instruction (and thus the IISA REG instruction) forms an index into load data queue **80** and the TA provides verification that the speculatively calculated target address was correct. Although implementing a load data queue such as that depicted in **Figure 3** may reduce access latency in some implementations, the improvement in access latency entails additional complexity in that store operations and exclusive access requests by other processors must be snooped against the load data queue to ensure correctness.

In another embodiment of the present invention, it may be desired to permit the PREFETCH instruction to be issued and executed as early as possible, but still constrain the PREFETCH instruction to be executed without utilizing speculative address operands. That is, when dispatching instructions, ISU **20** would still advance the PREFETCH instruction as far as possible in execution order with respect to the REGISTER instructions, but processor **10** would enforce register data dependencies so that PREFETCH instructions would always use correct

(i.e., non-speculative) register values when computing the prefetch address.

As has been described, the present invention provides an improved processor and method of performing load operations that translate UISA load operations into separately executable prefetch and register operations. Because performing the prefetch operation does not affect the architected state of a processor, the prefetch operation can be performed speculatively to mask data access latency, even in in-order execution machines. The register operation can thereafter be performed in-order to complete the load operation.

While the invention has been particularly shown and described with reference to a preferred embodiment, it will be understood by those skilled in the art that various changes in form and detail may be made therein without departing from the spirit and scope of the invention.